On evaluation of semantic-based search over XML document streams – Technical Report

Maciej Gawinecki
ICT School, Computer Science, XXIII cycle
DII, University of Modena and Reggio Emilia
Via Vignolese 905, 41100 Modena, Italy
maciej.gawinecki@unimore.it

ABSTRACT

Summarising, aggregating, and querying streams is a very interesting problem, one a lot of the big names (like Microsoft Research) are working on. In this report we are sharing with our experience on evaluation of semantic-based search over XML document streams. We present both theoretical and practical framework we developed, to compare behaviour of two heuristics, SLCA and XRank, on processing streamed DBLP data. Our report is a work in progress as it goes in many research directions and leaves many questions open. We are circulating it now for feedback.

1. INTRODUCTION

Data streams differ from conventionally stored model: (1) data elements in the stream arrive online, (2) system has no control over order in which data elements to be processed, (3) data streams are potentially unbounded in size (4) once an element from a data stream has been processed, it is discarded or archived. It cannot be retrieved easily unless it is stored in memory, which is small relative to the size of data streams [11]. The main interest in this domain is on evaluating structured queries over streams [10]. Traditionally, such streams compose data coming from various sources and arrive to query engine, which evaluates already registered queries over them. In modern environment new data sources can appear online with new structure. Without lack of knowledge about the document's structure in advance the user cannot register any structural query against new data source. The answer to this problem is keyword search approach. The user is relaxed from defining exact query, i.e. what are relations among the data she looks for and what results are to be returned (clause `SELECT` or `RETURN` in popular query languages), as these responsibilities are moved to the system.

Unfortunately, there is not so much work done on evaluating keyword search on XML document streams. In this report we address this problem and describe a simple implementation of the evaluation framework.

Precisely, we want to evaluate on how two heuristics, namely XRank and SLCA, behave on processing XML document streams. A stream can be generated by streaming persistent XML document and processed with a usage of SAX parser [6]. A query is defined as a set of query keywords and is evaluated over the flowing stream by one of the heuristics.

The evaluation process produces *real results*. To denote how objectively relevant to the query real results are, we construct precision/recall summary, where ideally relevant results are called *expected results*. The latter must be defined in advance in a certain explicit and precise way, so we were able to make comparisons automatically, especially for large sets of results. Therefore the same set of XML documents must be queried with a query which explicitly defines what result will be returned and what conditions this result have to satisfy. We propose to use XQuery language [9] to define such a query. Set of expected results is generated by executing the query with Berkeley DB XML database. Whole scenario constitutes effectiveness evaluation process and has been illustrated on Figure 1.

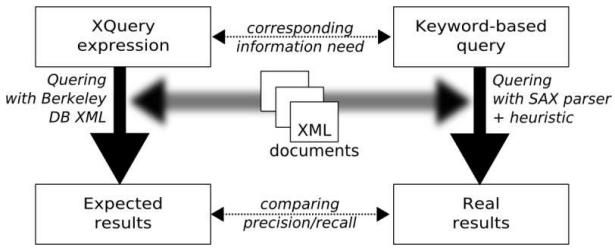

Figure 1: Schema of effectiveness evaluation process.

In the next section we present theoretical framework for evaluation. In the Section 3 we speak about writing test configurations and reading test results. In the Section 4 we describe practical architecture of the framework and additional tools. In section 5 we test our framework on DBLP data, analyse behaviour of both heuristics and show how this analysis can be supported by our tools. Finally in Section 6 we make conclusions and present open questions for further research.

2. THEORY

In this section we define a syntax of query a user can pose to the system. We also define two heuristics which the system can employ to evaluate query over the incoming XML document stream. We describe how the system identifies and demarcates results of the query and how it ranks them to generate ranking. We define also how both heuristics and ranking model can be applied to streaming data model in terms of algorithms.

2.1 Query model

We define here a query language which enables a user to explicitly specify constraints on the labels and/or textual content [16]. Thus, the query language we are presenting here allows us to define that we are looking for Wooldridge as an author of the publication, not as an editor. This way, we are applying more semantics to the simple keyword-based search.

Let us define now the form of a query.

DEFINITION 1. ***XML Semantic Search Query***. *Assume $a(n)$ (infinite) stream of documents $< d_i >$, $i = 1, \ldots$ Moreover, let $\{N_i\}$ be the set of XML nodes within document d_i. An XML semantic search query Q over this XML document stream is a list of query terms $(t_1, \ldots, t_m)$. Each query term is of the form: $l :: k$, $l ::$, $:: k$, or k, where l is a node label and k a keyword. A node n_i within document d_i satisfies a query term of the form:*

- *$l :: k$ – if n_i's label equals l and the tokenized textual content of n_i contains the word k.*
- *$l ::$ – if n_i's label equals l.*
- *$:: k$ – if the tokenized textual content of n_i contains the word k.*
- *k – if either n_i's label is k or the tokenized textual content of n_i contains the word k.*

*We focus on conjunctive keyword queries, where query terms are separated by a comma. For such a query only elements which satisfy **all** query terms are returned.*

*Each query term t_i results in a separate **input stream** IS_i, which contains all the nodes that satisfy the term. The answer to Q consists of a stream of document fragments $< df_i >$, $i = 1, \ldots$. Each such document fragment contains at least one tuple $(n_1, \ldots, n_m)$, $n_i \in IS_i$ with the additional constraint that the nodes in this tuple are **meaningfully related**.*

2.2 Search heuristics

There have been devised a number of heuristics defining when a group of nodes contains **meaningfully related** nodes. We employed two heuristics relaying on the notion of the *Lowest Common Ancestor* (LCA) of two nodes n_1 and n_2 [14]. First, assume that an XML semantic search query has produced two input streams IS_1 and IS_2:

DEFINITION 2. ***XRank heuristic*** *(from XRank system [12]) asserts that two nodes n_1 and n_2 that belong to the same XML document d_i, with $n_1 \in IS_1$ and $n_2 \in IS_2$, are meaningfully related if there are no nodes $n_3 \in IS_1$ and $n_4 \in IS_2$ that also belong to the same document d_i, such that $LCA(n_1, n_3)$ or $LCA(n_2, n_4)$ is descendant of $LCA(n_1, n_2)$.*

DEFINITION 3. *The **SLCA heuristic**[1] [18] states that two nodes n_1 and n_2 that belong to the same document d_i, with $n_1 \in IS_1$ and $n_2 \in IS_2$, are meaningfully related unless there exist two other nodes $n_3 \in IS_1$ and $n_4 \in IS_2$ that also belong to d_i such that the $LCA(n_3, n_4)$ is a descendant of the $LCA(n_1, n_2)$.*

The intuition behind that both heuristics is that smaller trees contain more meaningfully related nodes. However,

[1]SLCA – Smallest Lowest Common Ancestor.

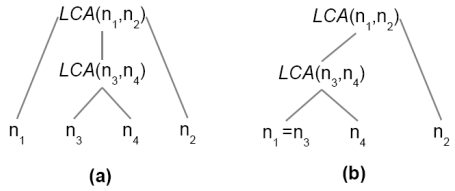

Figure 2: Example of meaningfully related nodes.

the difference between the SLCA and the XRank heuristics lays in that the first one disqualifies a pair of nodes n_1 and n_2 if there exists any other pair (n_3, n_4) with $LCA(n_3, n_4)$ being a descendant of $LCA(n_1, n_2)$, while XRank has the additional constraint that $n_1 = n_3$ or $n_2 = n_4$. For example on Figure 2 in case (a) the nodes n_1 and n_2 would be disqualified by SLCA heuristic, though they can be meaningfully related in terms of XRank ($n_1 \neq n_3$ and $n_2 \neq n_4$). In case (b) nodes n_1 and n_2 would be disqualified.

We have discussed the effectiveness of the relatedness heuristics for two input sets. To verify whether n ($n > 2$) nodes in a set S are meaningfully related, the SLCA and XRank heuristics employ the notion of LCA of the nodes in S (LCA_S). In this case, the nodes in S are related unless there exists a set S', such that $LCA_{S'}$ is a descendant of LCA_S [16].

2.3 Ranking results

Ranking of results present to what degree subsequent results are relevant to the query.

2.3.1 Returning query results

We want to decide which nodes belongs to the query result df_1. Definitions of both heuristics, XRank and SLCA imply that such a document fragment df_1: (1) contains at least one meaningfully related tuple of nodes, which does not belong to any other result and (2) is rooted at r, which is LCA node of these nodes.

The difference between both heuristics lays in the fact that for SLCA heuristics if a node n is identified as root of a query result, then none of n's ancestors can be root of another query answer. As this is not the case for XRank heuristic, therefore we make another assumption about its result: a result contains the set of elements that contain at least one occurrence of each query term, after *excluding* the occurrences of the keywords in subelements, that already contain all of the query keywords [12].

These characteristics can be easily observed in algorithms of SLCA and XRank heuristics and also in the Figure 3. The picture presents situation where in XML document there exists two pairs of meaningfully related nodes, namely n_1,n_2 and n_3,n_4. XRank heuristic (case (b)) returns two separate results rooted at LCAs of both pairs of nodes, while SLCA heuristic (case (a)) returns only a result containing nodes n_3,n_4, because nodes n_1,n_2 belong to the tree rooted at $LCA(n_1, n_1)$ node, which is ancestor of $LCA(n_3, n_4)$ node.

2.3.2 Ranking model

In [12] authors proposed XRank system with ranking function for XML documents. We want to utilize their model, but we made several simplification about it, because the nature of our problem is slightly different.

Particularly their function works on persistent and indexed documents, while we are interested in querying docu-

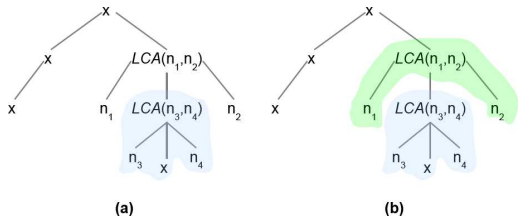

Figure 3: Document fragments belonging to a query results set.

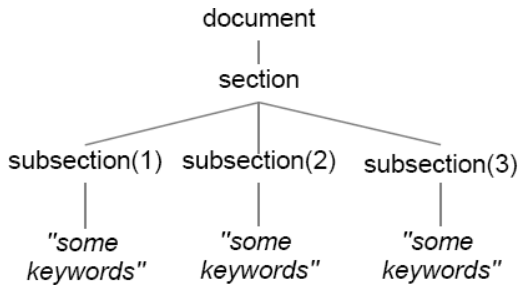

Figure 4: Example of results specificity.

ment streams. The streams can be potentially unbounded in size and new elements arrive online and are processed only once. Therefore it is not possible to know in advance a number of elements or word frequency vector of the document. Obtaining at least partial knowledge about these factors requires finding a compromise between memory boundaries and number of document elements archived when processing the stream.

Please note also, that authors of XRank system are interested in processing not only XML documents, but also (X)HTML documents, which are loosely structured. On the contrary, we want to process only highly structured XML documents.

Another simplification is that, we would like our framework to work only with XML documents, which can be modelled as a tree, i.e. we are not considering hyperlinked structure of XML documents.

Now we can outline what precisely are properties for simplified ranking function over highly-structured XML document streams.

1. *Result specificity.* The ranking function should rank more specific results higher than less specific results. For instance, on Figure 4 a *subsection(1)* result (which means that all query keywords are in the same subsection) should be ranked higher than a *section* result (which means that the query keywords occur in different subsections). Moreover, from assumption about results of XRank heuristic search (see Section 2.3.1) we know that results df_1, df_2 can be rooted at nodes n_1,n_2, respectively, where n_1 is an ancestor of n_2, but df_1 cannot be considered as a tree containing subtree df_2. Therefore the rank of df_2 cannot be *aggregated* into the rank of df_1.

2. *No keyword proximity.* Keyword proximity is a continuous function which says how much are occurrences of query keywords distributed in the returned document fragment. However, as we said, we are interested in processing only highly structured XML data (with only some flavour of natural language text), where the distance between query keywords may not always be an important factor. Therefore, the keyword proximity function can be set to be always 1 [12]. Moreover, streamed documents are processed by a search heuristics usually within a single scan. Therefore, in case of measuring keyword proximity, it would be necessary to cache position of matching keyword occurrences. In the worst case it would result in resource complexity of $O(n)$, where n is the number of keywords in the processed document. The answer whether this complexity can be compensated by better quality results is a problem for future research.

3. *No hyperlink awareness.* As we are not considering links in XML documents, then we do not have any objective (i.e. without respect to a user query) knowledge about ranks of particular elements in a document stream. Therefore we assume that objective rank of a single element is 1.

2.4 Ranking function – definition

According to the present requirements of ranking function now we define it more formally.

2.4.1 Ranking with respect to one keyword

Let r be the root node the document fragment df, returned as a result of the query $Q = (t_1, \ldots, t_m)$. We want to define the ranking of r with respect to one occurrence of query keyword t_i: $rank(r, t_i)$. From definitions of both heuristics (Section 2.2) we know that there exists at least one occurrence of this keyword in df. Let us assume that n_{t_i} is a node directly containing such an occurrence. From [12] we define *rank* in the following way.

$$\begin{aligned} rank(r, t_i) &= rank(t, n_{t_i}) \\ &= ElementRank(r) \cdot decay^h \\ &= decay^h \end{aligned} \tag{1}$$

where h is a path distance[2] between nodes: r and the node n_{t_i}. Intuitively, the rank of r with respect to a keyword t_i is $ElemRank(r)$ scaled appropriately to account for the specificity of the result. For each edge on the path between nodes r and n_{t_i}, the rank is scaled down by the factor *decay* for each level, where *decay* is a parameter that can be set to a value in the range 0 to 1. In our case $decay = 0.4$.

$ElemRank(r)$ denotes objective (independent to the query) ranking of node r, but as we discussed already, we do note have such a knowledge and thus for each r the function $ElemRank(r)$ returns 1.

Of course, document df can contain more than one occurrence of the query keyword t_i. In this case for each occurrence, we compute *rank* separately and then aggregate rankings with some aggregation function f (here we fixed $f = \max$). Formally, having k occurrences of query keyword t_i, we will have k nodes[3] containing these occurrences,

[2]Path distance can be easily computed, because $h = depth(n_{t_i}) - depth(r)$. The function $depth(x)$ returns a path distance from x node to the root node of whole XML document.

[3]Here we assumed, that a single node does not directly con-

namely: $n^1_{t_i}, \ldots, n^k_{t_i}$. By $n^j_{t_i}$ we denote a node containing j-th occurrence of the query keyword t_i. Then we can compute k rankings of these occurrences, namely $r_1, \ldots, r_k$. Finally, we denote ranking of r in respect to all occurrences of a keyword t_i by $rank^*(r, t_i)$:

$$\begin{aligned} rank^*(r, t_i) &= f(r_1, \ldots, r_k) \\ &= \max_{j=1\ldots k} r_j \end{aligned} \tag{2}$$

2.4.2 Overall Ranking

The overall ranking of a result element r for query $Q = (t_1, \ldots, t_m)$ is computed as follows:

$$\begin{aligned} Rank(r, Q) &= \left(\sum_{t_i \in Q} rank^*(r, t_i) \right) \cdot p(r, t_1, \ldots, t_m) \\ &= \sum_{t_i \in Q} rank^*(r, t_i) \end{aligned} \tag{3}$$

The function p is a keyword proximity function which we assumed to be always set to be 1.

2.5 Computing effectiveness

Effectiveness is a measure of the ability of the system to satisfy the user query in terms of relevance of returned results [17]. Here we concentrate on measuring effectiveness by precision and recall. The output of retrieval process in our system is a list of results sorted descendingly in respect to their ranks. For each position in the ranking – λ – we can calculate corresponding precision-recall value, here denoted by ordered pair (R_λ, P_λ). The set of ordered pairs makes up the precision-recall curve.

2.5.1 Interpolation of precision/recall values

The precision-recall curve is usually based on 11 standard recall levels, which are: 0.0, 0.1, 0.2, ..., 1.0. However, it often happens that no unique precision value corresponds exactly to them, and then it becomes necessary to interpolate. For this purpose we adopted *conservative interpolation* method [17], described here shortly.

Let us obtain the set of observed precision/recall points in the ranking. To do this, we specify such a subset of the parameters λ, that (R_θ, P_θ) is an observed point if θ corresponds to a value of λ at which an increase in recall is produced. Let $G_s = (R_{\theta_s}, P_{\theta_s})$ be the set of observed points for a single user request (query) s. To interpolate between any two points we define:

$$P_s(R) = \{supP : R' \geq R, s.t.(R', P) \in G_s\} \tag{4}$$

2.6 Algorithms

In order to support keyword search over XML document streams we need to apply respective algorithms, which process arriving documents within a single scan. For this purpose we employed algorithms for XRank and SLCA heuristics from the work [13]. We present these algorithms in pseudocode, with our modifications highlighted in gray background. Here we briefly describe general idea of the solution proposed in mentioned work and precisely describe our modifications.

tain more than one occurrence of the given keyword or even if it contains it is not counted. This can be reasonable assumption for highly structured documents.

Algorithm 1 Evaluate Search Heuristic

```
1.  {Input: Set of Query Terms Q_t}
2.  {       Stream of XML documents S_d}
3.  {Output: Set of result XML nodes R_t}

4.  R_t := ∅
5.  for d ∈ S_d do
6.    s.clear() {initialize node stack}
7.    R_t := R_t ∪ SAX_Parse(d, Q_t, s, R_t)
8.  end for
9.  return R_t
```

Algorithm 2 Evaluate XRank Heuristic – Start Tag

```
1.  {Input: Accessed Document Node n}
2.  {       Set of Query Terms Q_t}
3.  {       Node Stack s}
4.  {       Set of Nodes Cache}

5.  s_n.label := n.label {create new stack node}
6.  s_n.term_instances := ∅
7.  if ∃t ::∈ Q_t : t == n.label then
8.    s_n.term_instances(t ::) := FOUND
9.    Cache := Cache ∪ s_n
10. end if
11. if ∃k ∈ Q_t : k == n.label then
12.   s_n.term_instances(k) := FOUND
13.   Cache := Cache ∪ s_n
14. end if
15. s.push(s_n)
```

2.6.1 Algorithms for heuristics

General search process for the given search query Q_t and sequential stream of documents S_d has been illustrated in Algorithm 1. Algorithm parses each document d from the stream S_d and add results back to the set R_t. Each document is parsed by SAX parser [6], which returns the following three types of events: $startElement(tag)$, $characters(text)$, $endElement(tag)$. Algorithms handle these events through the SAX call back methods.

SAX parser scans a document in a in-order fashion and thus each node is visited (*startElement* event) before visiting all its descendants. After visiting them or, in case of a leaf node, reading text value of the node, *endElement* event is triggered by the parser. For both events, *startElement* and *endElement*, the algorithm checks whether currently processed node label or value matches one of the query terms. *Node Stack* structure is used to store all nodes with opened tags. Each node in the stack has a corresponding *Bitmap* structure denoting whether the node or one of its descendants matches one of the query terms. When the *Bitmap* is complete, i.e. it contains all matches then it means that the corresponding node is a root of a result and all query terms has been matched by subset of it descendant nodes. This algorithm works similarly for both heuristics. However, algorithm for SLCA heuristic takes into account the fact, that when a result is found, none of its ancestors can be a result (*CAN_BE_SLCA* flag).

Algorithm 3 Evaluate XRank Heuristic – End Tag

```
1.  {Input: Set of result XML nodes R_t}
2.  {       Set of Nodes Cache }

3.  r_n := s.pop()
4.  if r_n.term_instances == COMPLETE then
5.    rank(r_n, Cache)
6.    R_t := R_t ∪ {r_n}
7.  else
8.    if not s == EMPTY then
9.      for t_k ∈ r_n.term_instances.keys do
10.       if r_n.term_instances(t_k) == FOUND then
11.         s.top().term_instances(t_k) := FOUND
12.       end if
13.     end for
14.   end if
15. end if
```

Algorithm 4 Evaluate Search Heuristic – XML Text

```
1.  {Input: Textual Content text }
2.  {       Node Stack s }
3.  {       Set of Nodes Cache }

4.  for word ∈ tokenize(text) do
5.    if ∃ :: k ∈ Q_t : k == word then
6.      s.top().term_instances(:: k) := FOUND
7.      Cache := Cache ∪ s_n
8.    end if
9.    if ∃k ∈ Q_t : k == word then
10.     s.top().term_instances(k) := FOUND
11.     Cache := Cache ∪ s_n
12.   end if
13.   if ∃t :: k ∈ Q_t :
          (t == s.top().label AND k == word) then
14.     s.top().term_instances(t :: k) := FOUND
15.     Cache := Cache ∪ s_n
16.   end if
17. end for
```

Algorithm 5 Evaluate SLCA Heuristic – Start Tag

```
1.  {Input: Accessed Document Node n}
2.  {       Set of Query Terms Q_t}
3.  {       Node Stack s}
4.  {       Set of Nodes Cache }

5.  s_n.label := n.label {create new stack node}
6.  s_n.CAN_BE_SLCA := TRUE
7.  s_n.term_instances := ∅
8.  if ∃t ::∈ Q_t : t == n.label then
9.    s_n.term_instances(t ::) := FOUND
10.   Cache := Cache ∪ s_n
11. end if
12. if ∃k ∈ Q_t : k == n.label then
13.   s_n.term_instances(k) := FOUND
14.   Cache := Cache ∪ s_n
15. end if
16. s.push(s_n)
```

2.6.2 Identifying results

In terms of the query the user provides only keyword terms which is she looking for, but does not define what part of relevant document fragments are to be returned, as it is usually done in query languages (clause `SELECT` or `RETURN`). She may be looking for wider context in which the meaningfully related nodes appear (document fragment rooted in LCA of these nodes), but presentation of the context, especially modelled as a deep XML tree, is challenging user interface problem which we do not concern in this report. Instead, we present unique way to identify the root of a result and meaningfully related nodes within it. Moreover, unique identification of nodes in a XML document is necessary also during effectiveness evaluation of retrieval algorithms, so we could automate process of checking whether returned result is relevant to the expected one.

Precisely, for uniqueness identification it is not enough to define a document within which a node is placed and a path leading to it from the root node of the document, because the path can ambiguous, i.e. it does not distinguish sibling nodes. To distinguish sibling nodes we use a notion of numeric predicate from XPath language [8]. For example the following position description:

label:	author
document:	`file:etc/data/dblp/dblp-example.xml`
path:	/dblp[1]/incollection[3]/author[3]
column:	11
row:	34

says the node can be found in local file `etc/data/dblp/dblp-example.xml`. To find the node in the document we need to follow the path: first we choose 1^{st} *dblp* node, then 3^{rd} *incollection* node and eventually – 3^{rd} *author* node. This notation is independent to document format, i.e. does not depend on new lines and intendation characters. Moreover, the order of siblings is preserved also after loading XML document into one of XML databases (e.g. Berkeley DB XML) and can be used further to query the database for the node with a respective XPath expression.

However, when working on the XML document with classical text editor, the task of following along the location path can be quite uphill for bigger documents. For this purpose we provide also position of the node in terms of column and row (line) numbers. Column and row of a node during parsing can be easily obtained from SAX locator object, which associates returned SAX events with a document location. Precisely SAX locator returns the position where the currently processed document event ends [6]. However, there could be a difference between position returned by SAX locator and one provided by text editors. We did not observe this problem during working with SAX locator, but the documentation says explicitly, that lack of correspondence can occur when *"lines contain combining character sequences, wide characters, surrogate pairs, or bi-directional text"*. Moreover, *"the return value from the method [of locator] is intended only as an approximation for the sake of diagnostics; it is not intended to provide sufficient information to edit the character content of the original XML document"* [6].

2.6.3 Algorithm for ranking

Algorithm 6 Evaluate SLCA Heuristic – End Tag

```
1.  {Input: Set of result XML nodes R_t}
2.  {        Set of Nodes Cache }

3.  r_n := s.pop()
4.  if r_n.CAN_BE_SLCA == FALSE then
5.    if not s == EMPTY then
6.      s.top().CAN_BE_SLCA := FALSE
7.    end if
8.  else
9.    if r_n.term_instances == COMPLETE then
10.     rank(r_n, Cache)
11.     R_t := R_t ∪ {r_n}
12.     s.top().CAN_BE_SLCA := FALSE
13.   else
14.     if not s == EMPTY then
15.       for t_k ∈ r_n.term_instances.keys do
16.         if r_n.term_instances(t_k) == FOUND then
17.           s.top().term_instances(t_k) := FOUND
18.         end if
19.       end for
20.     end if
21.   end if
22. end if
```

Algorithm 7 Rank Result Node

```
1.  {Input: Result node r_n}
2.  {        Set of Nodes Cache }

3.  M := ∅ {matching nodes}
4.  for c ∈ Cache do
5.    if c.position ≥ r_n.position then
6.      M := M ∪ {c}
7.      Cache := Cache − {c}
8.    end if
9.  end for
10. r_n.rank := computeRank(r_n, M)
```

In section 2.3.2 we defined a model according to which we want to rank query results. To rank a single result (document fragment) we need to know the root of the result and all nodes satisfying query terms. This information can be easily collected during parsing of XML document stream. Therefore we could effortlessly enhance existing retrieval algorithms with ranking feature by introducing a set $Cache$, where all matching nodes are added. For example in the line 9 of Algorithm 2, the node which matches a query term of $t ::$ form is added to the $Cache$.

When a single result *df* is found and its root node is returned, then we have all necessary information to rank it, i.e. descendant nodes of the root has been visited, because SAX parser search an XML tree with Depth-First-Search algorithm. The result is ranked by the function $rank(r_n, Cache)$ called in Algorithm 3 and Algorithm 6, and defined in Algorithm 7.

However, let us observe, that the $Cache$ set can contain nodes not only belonging to the currently ranked result *df*, but also to other document fragment being a potential result. The case of XRank heuristic is presented on Figure 3 (b). Here, when the document fragment rooted at $LCA(n_3, n_4)$ node is returned, the $Cache$ set contain not only n_3, n_4, but also nodes n_1, n_2 which belongs to the fragment rooted at $LCA(n_1, n_2)$ node. To separate nodes in $Cache$ belonging to different document fragments we can use the *in-order*, according to which SAX parser visits subsequent nodes of XML document. The order can be formally defined as follows. For any two nodes n_1 and n_2 with defined positions:

$$position(n_1) = (col_1, row_1)$$
$$position(n_2) = (col_2, row_2)$$

we can say that the node n_1 lays *before* the node n_2 iff:

$$row_1 < row_2 \vee (row_1 = row_2 \wedge col_1 < col_2).$$

We can observe, that in respect to this order the nodes belonging to the currently ranked document fragment *df* are equal or after the root node of *df*. Therefore, in Algorithm 7, we first remove respective nodes from $Cache$ and add them to the set M, over which the function *computeRank* computes the rank according to the equations given in the section 2.3.2.

3. TEST CONFIGURATION AND LOGS

We assumed that typical use case of the application consists of the following steps:

1. load *test configuration* from a file,
2. execute retrieval process and measure its execution time (efficiency evaluation),
3. compare results found with expected results and construct precision/recall summary (effectiveness evaluation),
4. save the results and evaluation results to a *test log*.

Naturally we uses XML to describe both *test configuration* and *test log* files. This allows us to fully automate process of experimenting with the application. Precisely, test configuration file need to be defined in respect to *Test Collection XML schema*[4]. Both test configuration and log can be easily read and navigated in any popular Internet browser[5][6] (Microsoft IE recommended). Please see Figure 5 for an example of this feature.

3.1 Test configuration

An example of such a test configuration file has been presented on Figure 6. We can observe that the root element – *testCollection* – contains *testCase* elements. Each *testCase* defines a *searchProfile* – separate search task to evaluate. The same search task can be tested under diverse conditions (e.g. with various search heuristics), defined within

[4]The schema has been described in `etc/tests/testCollection.xsd` file. Extensive HTML documentation of this schema can be found in `doc/testCollection/index.html` file. Here we only give a brief insight into it.

[5]For this purpose we provided XSL stylesheet (`etc/tests/xml-to-html.xsl` file) transforming test descriptions and results (logs) into HTML form.

[6]For log files we decided not to include results below 100^{th} position in the ranking in a test log, as this resulted in log of vast size, e.g. of 4 MB size for 0.2 scale factor of XMark data, which is a CPU killer for a browser to render it.

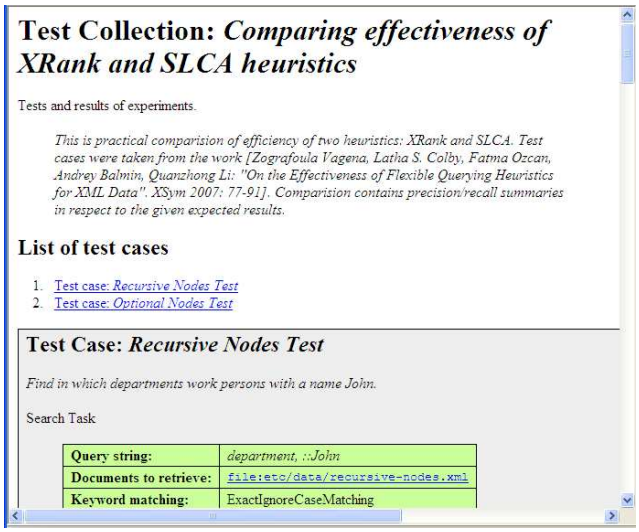

Figure 5: Example of a test collection presented in the Internet browser.

```
<?xml version="1.0" encoding="UTF-8"
  standalone="yes"?>
<?xml-stylesheet xtype="text/xsl"
  href="xml-to-html.xsl" ?>
<ns2:testCollection
  name="A node matching"
  xmlns:ns2="http://unimore.it/IR/search/domain#">
 <description>
  It shows how heuristics...
 </description>
 <testCase name="Recursive Nodes Test"
   skip="false">
   <searchProfile>...</searchProfile>
   <experiments>...</experiments>
 </testCase>
 ...
</testCollection>
```

Figure 6: Example of *test configuration* file.

set of *experiments*. The schema defines *searchProfile* and *experiment* as XML complex types: *SearchProfile* and *Experiment*, accordingly.

3.1.1 Search profile definition

SearchProfile type (see Figure 7) describes mainly *SearchTask* in terms of

- *query*, defined with the syntax proposed in the section 2.1,
- *document*s, i.e. list of URI address of documents to retrieve
- *keywordMatching*, policy saying how the query keyword will be matched with label or word contained in the XML node; as for now matching can be either exact with case insensitivity or based on substrings (two strings matches of one is a substring of another).

SearchProfile defines also a list of *expectedResults* to evaluate effectiveness. It contains also XQuery expression describing how these results has been generated. An excerpt describing search profile is presented on Figure 9.

Here we are posing <*author::Jurgen,author:Daniel*> query against local file `etc/data/dblp/dblp.xml`. Compared strings have to be equal (but without regard to the character case) to be matched. The expected result has to be rooted at node identified by XPath expression */dblp[1]/incollection[2]*. This result has been generated on the based of the given XQuery expression.

3.1.2 Experiment definition

Experiment complex type (see Figure 8) defines the conditions under which the search task will be performed. It points out:

- *evaluator*, here heuristic, that will be used to evaluate query,
- *rankingModel*, which will be used (as for now only XRank ranking model is available),
- *asynchronous* mode, i.e. whether the experiment is to be executed in asynchronous way (this feature is described in section 5.3.1).

3.2 Test log

Test log is a XML document generated as a result of execution experiments defined in particular test configuration file. Please note, that each experiment can be executed several times.

3.2.1 Experiment execution definition

Each *execution* is identified by execution time (*executed* parameter). Efficiency is measured with various precision: (*elapsedSeconds* and *elapsedNanoseconds* parameters). Each execution contains also a set of *results* and a precision versus recall summaries: *extendedSummary* (for all observation points generating increace of recall) and *standardSummary* (interpolated for 11 standard recall levels).

On Figure 10 we illustrate single experiment together with one execution. The experiment evaluates a query with XRank heuristic and XRank ranking model in synchronous mode. It has been executed once: on September 2, 2008 at 20:33. The execution returned one relevant result ranked with value 1.0 and rooted at node identified by */dblp[1]/incollection[2]* XPath expression. One of meaningfully related nodes of this result is identified by */dblp[1]/incollection[2]/author[1]* XPath expression and matches both query terms from the query. The effectiveness summary tells us that general precision is 33% (probably 1 of 3 answers where relevant). However, standard summary tells us that for all standard recall levels single precision values were at level of 100%.

4. IMPLEMENTATION

We implemented the framework in Java 1.6 as an Eclipse project. Compilation and launching can be automatically performed with Ant targets[7]. In this sections we present

[7]For the details about installation and launching particular tasks please refer to the `README.TXT` file in the root directory of the project.

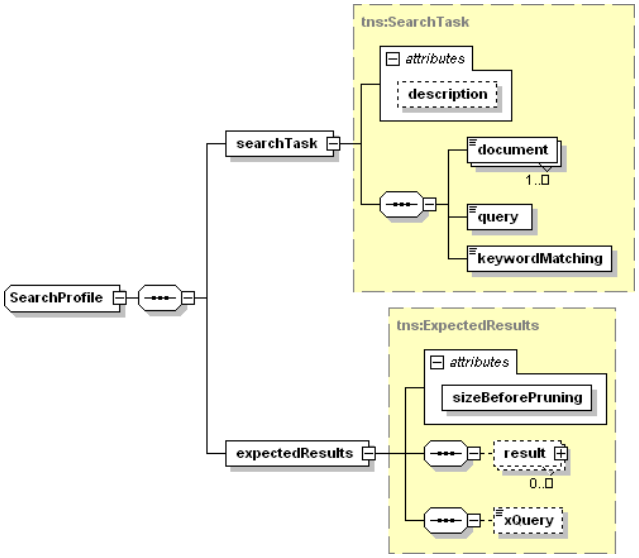

Figure 7: *SearchProfile* complex type.

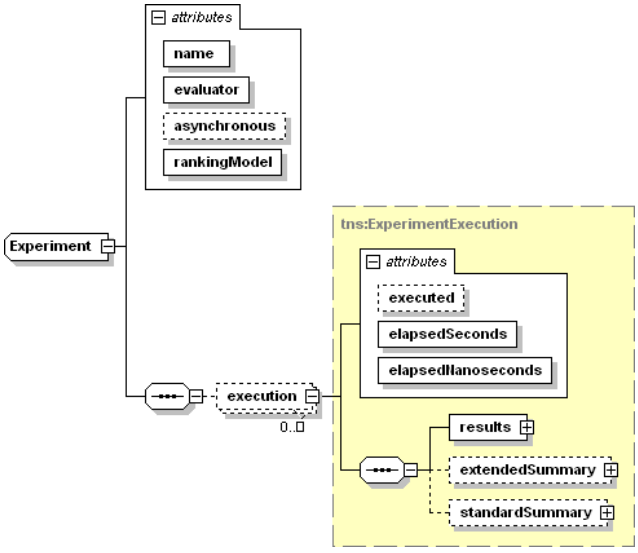

Figure 8: *Experiment* complex type.

core architecture of the framework and functionality of additional tools supporting evaluation process.

4.1 Evaluation framework

Design of retrieval application reflects the typical use case scenario described in Section 3. Tests can be launched from command line:

```
java it.unimore.ir.search.benchmark.LaunchTest
  TEST_CONF_FILE TEST_LOG_FILE
```

where `LaunchTest` class reads in test configuration file (*TEST_CONF_FILE*), performs defined tests and writes results back to test log file (*TEST_LOG_FILE*). We use JAXB (Java Architecture for XML Binding) [3, 4] to unmarshal test configuration demarcated in XML into Java objects and marshal results back into XML log.

Description of how tests are performed by subsequent components in the implementation has been presented on Figure 11. Arrows depict control flow among components[8]. For each experiment in each test case we create an instance of `ExperimentRuntime` class. The instance is constructed according to the parameters of search task of the test case and parameters of the experiment. Therefore it creates `evaluator` object (respective to the defined heuristic) and `rankinizer` (respective to the defined ranking model). Next, each document listed in the search task is parsed by an instance of the implementation of `SAXParser` interface[9]. For each event occurred during parsing `SAXParser` invokes a respective callback method of `SearchHandler` class. The latter is responsible for remembering current path in the processed XML document, and caching label and contained text of the currently processed XML node, so the `evaluator` object could evaluate the query over them. As the `evaluator` object can be an instance either of `SLCAHeuristicQueryEvaluator` class or `XRankHeuristicQueryEvaluator` class, the further behaviour is direct implementation of respective heuristic algorithms presented in Section 2.6.1. Generally, both heuristics check which of a query terms in the query matches the cached label/word. String matching is done by `comparator` object according to the keyword matching politics defined in the search task. When there is a match, a node is marked by the heuristics and also cached by `rankinizer` object. Finally, when the result is found, it is being ranked by the `rankinizer`.

The result of `ExperimentRuntime` work is a list of query results (instances of `Result` class) ordered descendingly according to their ranks. These results are compared with expected results from the search task by `EffectivenessCalculator` class. The effect of the comparison is precision/recall summary for the given search task, expected results and experiment conditions.

4.2 Tools

In addition to the core of the framework, we developed also three tools supporting work with XML documents, test configurations and logs. Sources of these tools can be found in `it.unimore.ir.search.tools` package.

4.2.1 Relevant Answers Generator

In our case the test collection should contain: (a) a volume of XML data, (b) a set of testing queries demarcated in some XML query language, (d) a set of the same queries but expressed in the syntax accepted by our engine (see Section 2.1) and (d) sets of expected answers, relevant to these queries. Well known TREC test collection does not consider XML documents at all, while DBLP and Wikipedia XML Corpus collections contains only pure data. Therefore we decided to prepare our own queries and relevant answers. While constructing queries is straightforward task, the process of discovering relevant answers is uphill and thus should be automated. For this purpose we developed a tool that generates a set of relevant answers for the given XQuery [9] expression and list of documents. A set of relevant answers must be demarcated in a form understandable by the evaluation framework. Precisely, the tool must be able to create a

[8]JavaDoc API documentation of these components can be found in `doc/api/` directory.
[9]from Apache Xerces Java Parser library [1].

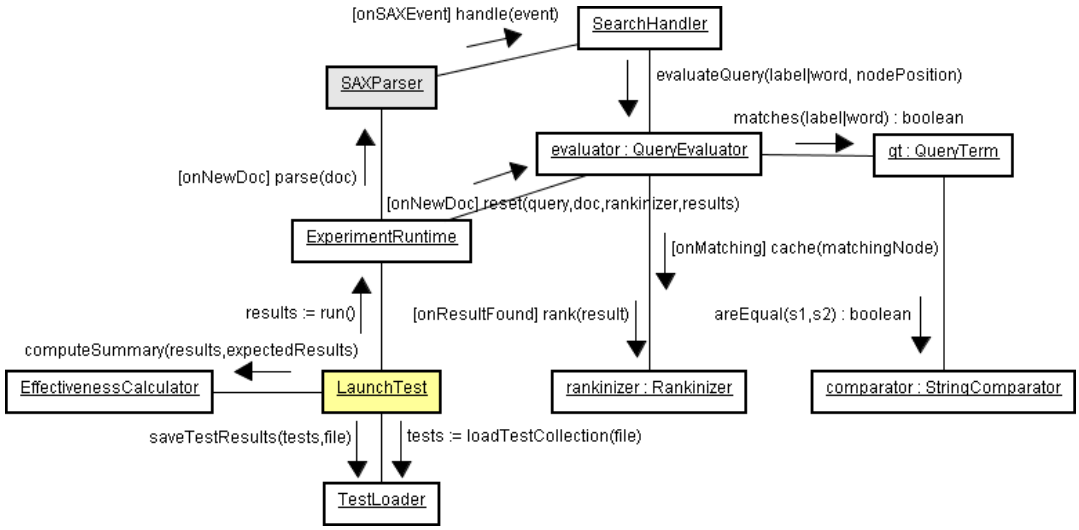

Figure 11: Collaboration diagram of the retrieval application.

generic XPath expression for a specific node reference representing a single answer, as it was described in Section 2.6.2. Therefore, for the given command:

```
java it.unimore.ir.search.tools.FindRelevantAnswers
  TEST_CONF_FILE TEST_CONF_FILE_UPDATED
```

the tool reads in all test cases from the `TEST_CONF_FILE`. Each test case is processed separately. For the XQuery expression given in *expectedResults* element it executes the query on a set of documents defined for the test case and update *expectedResults* element with a set of found results, referenced by XPath expressions. Finally, all processed and updated test cases are written back to the `TEST_CONF_FILE_UPDATED`.

The tool uses Oracle Berkeley DB XML database [5] to query node storage container with XML documents[10].

4.2.2 Data Analyzer

For the given set of XML documents the tool generates CSV file (possibly understood by popular spreadsheet applications) describing characteristics of the documents in terms of, namely:

- document size (in MBs),
- number of XML nodes,
- maximal depth,
- number of distinct label paths,
- number of keywords values of XML nodes.

The tool uses SAX parser.

An example of such an analysis is shown on Table 1. It has been constructed for 11 documents generated by XMark XML data generator [7] and DBLP data [2]. We varied the size of the XMark documents from 14 MB to 446 MB, according to the scale factor range of XMark – from 0.1 to 1.0 and additionally – 3.0. The latter resulted in data of size comparable to DBLP data size. Please note also, that such size of the documents comes from pretty printing of them. Although, our experience shows that new lines and indentation significantly slow down parsing and can be avoided for testing purposes.

[10]Please remember the document must be put into the container (e.g. with Berkeley DB XML shell) before querying.

4.2.3 Result Analyzer

For the given set of test logs the tool generates CSV file (possibly understood by popular spreadsheet applications) describing average results for all experiments of each test case, namely:

- name of a test case,
- name of an experiment,
- list of documents separate with comma,
- a query,
- a name of an experiment,
- a number of real results (this number does not vary among executions),
- average time for all executions of the given experiment (in seconds).

The tool uses SAX parser.

5. TEST CASES

We would like to compare efficiency and effectiveness of querying XML document streams for both heuristics: SLCA and XRank. Differences in effectiveness between both heuristics results from structural differences of documents they process and has been observed on various, structurally rich XML documents by Vagena et al. [16]. We show here which of these differences can be observed on DBLP data and which cannot be. Finally, we provide practical evaluation of our theoretical considerations.

Document	Size (MB)	Nodes	Depth	Label paths	Keywords
XMark-0.1	14	167,865	12	502	590,918
XMark-0.2	29	336,244	12	514	1,182,657
XMark-0.3	44	501,498	12	514	1,768,441
XMark-0.4	59	667,243	12	514	2,358,860
XMark-0.5	73	832,911	12	514	2,930,488
XMark-0.6	89	1,003,441	12	514	3,549,469
XMark-0.7	104	1,172,640	12	514	4,145,778
XMark-0.8	118	1,337,383	12	514	4,715,038
XMark-0.9	133	1,504,685	12	514	5,304,714
XMark-1.0	148	1,666,315	12	514	5,877,272
XMark-3.0	446	5,010,250	12	514	17,678,441
DBLP	443	10,891,042	6	142	36,793,029

Table 1: Characteristics of retrieved documents.

Ant task	Description	Input file	Output file
Asynchronous execution			
test-dblp-sync	Search local DBLP synchronously	`dblpTest.xml`	`dblpTest-log.xml`
test-dblp-async	Search local DBLP asynchronously	`dblpAsyncTest.xml`	`dblpAsyncTest-log.xml`
test-dblp-online-sync	Search online DBLP synchronously	`onlineDblpTest.xml`	`onlineDblpTest-log.xml`
test-dblp-online-async	Search online DBLP asynchronously	`onlineDblpAsyncTest.xml`	`onlineDblpAsyncTest-log.xml`
Sibling relationships and optional elements (main test)			
prepare-main	Prepare relevant answers	`mainTest.xml`	`mainTest-updated.xml`
test-main	Launch main test for DBLP	`mainTest-updated.xml`	`mainTest-log.xml`
retest-main	Repeat main test 3 times	`mainTest-log.xml`	`mainTest-log.xml`
analyze-main-result	Analyze results of main test	`mainTest-log.xml`	standard output
One element satisfying more than one query term			
prepare-combination-1	Prepare relevant answers 1	`combineTest1.xml`	`combineTest1-updated.xml`
test-combination-1	Launch test 1	`combineTest1-updated.xml`	`combineTest1-log.xml`
prepare-combination-2	Prepare relevant answers 2	`combineTest2.xml`	`combineTest2-updated.xml`
test-combination-2	Launch test 2	`combineTest2-updated.xml`	`combineTest2-log.xml`
General analysis			
analyze-data	Analyze XMark and DBLP	XMark and DBLP files	standard output

Table 2: Ant targets for described test cases.

5.1 Theoretical comparison

DBLP data contain information about publications demarcated in XML. Below we present the complete excerpt of DBLP data:

```
<inproceedings mdate="2007-12-13"
    key="conf/sacrypt/GorlaPS07">
  <author>Elisa Gorla</author>
  <author>Christoph Puttmann</author>
  <author>Jamshid Shokrollahi</author>
  <title>Explicit Formulas for Efficient
    Multiplication in \mathbb<i>F</i><sub>3<sup>6
    <i>m</i></sup></sub>.</title>
  <pages>173-183</pages>
  <year>2007</year>
  <crossref>conf/sacrypt/2007</crossref>
  <booktitle>Selected Areas in Cryptography
    </booktitle>
  <ee>http://dx.doi.org/10.1007/
    978-3-540-77360-3_12</ee>
  <url>db/conf/sacrypt/sacrypt2007.html
    #GorlaPS07</url>
</inproceedings>
```

Maximal length of XML path in DBLP data is 6 (according to the Table 1), where only three first elements provide certain semantic meaning, like in the following path (taken from the above example):

```
/dblp/inproceedings/title/sub/sup/i
```

The rest of the path is always dedicated for HTML formatting elements (`sup`, `i`, `sub` etc.). Moreover, the first element in the path is a root node of the document, thus effective number of positions in the path, on which elements may vary, is 2. Therefore we can observe that DBLP is structurally shallow.

According to the nomenclature used in the mentioned work [16], we analysed four characteristics of search heuristics on the base of DBLP data, namely: recursive nodes, nested nodes, sibling relationships and optional nodes. We also analysed one new case not the described in the work [16], namely: element satisfying more than one query term.

5.1.1 Recursive and nested elements

SLCA gives false negatives for both *recursive* and *nested elements* [16]. An element is called recursive if it appears more then once on the same path from a root to a leaf (*document* element in Figure 12a). On the contrary, when tuple of meaningfully related elements has LCA, which is an ancestor

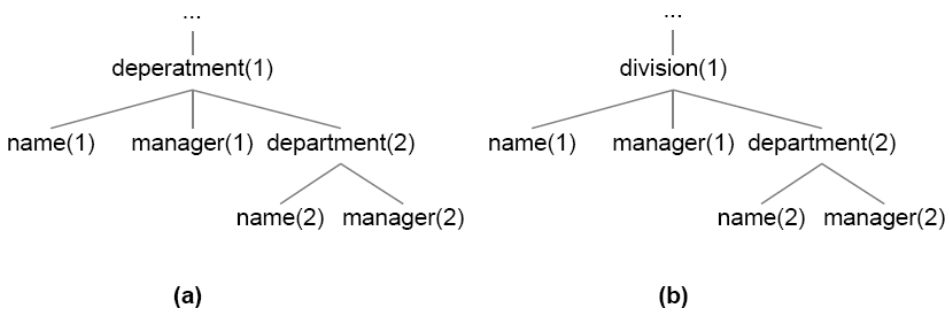

Figure 12: Examples of (a) recursive elements and (b) nested elements.

```
<searchProfile>
  <searchTask description="Find publication...">
      <document>file:etc/data/dblp/dblp.xml
        </document>
      <query>author::Jurgen,author::Daniel</query>
      <keywordMatching>ExactIgnoreCaseMatching
        </keywordMatching>
  </searchTask>
  <expectedResults>
    <result>
      <root label="incollection">
         <position path="/dblp[1]/incollection[2]"
           document=
             "file:etc/data/dblp/dblp.xml"/>
      </root>
    </result>
    <xQuery>
      for $x in collection("dblp.dbxml")/dblp/*
      for $y in $x/author
      for $z in $x/author
      where
      functx:contains-word($y,"Jurgen")
      and functx:contains-word($z,"Daniel")
      and $y != $z
      return $x
    </xQuery>
  </expectedResults>
</searchProfile>
```

Figure 9: Example of *searchProfile* section in *test collection* file.

of LCA of another tuple of meaningfully related elements, then we call these elements nested (e.g. *name* and *manager* elements in Figure 12b).

As we observe, these characteristics occur in documents of minimal depth of 3, while DBLP data have effective depth 2 and thus differences between heuristics cannot be noticed in this case.

5.1.2 One element satisfying more than one query term

Let us suppose we are looking for publications written by Jurgen and Daniel, but not by someone who is called Daniel Jurgen or Jurgen Daniel. This question can be easily expressed with XQuery in the following way:

```
for $x in collection("dblp-fiction.dbxml")/dblp/*
  for $y in $x/author
  for $z in $x/author
where
  functx:contains-word($y,"Jurgen")
  and functx:contains-word($z,"Daniel")
  and $y != $z
return $x
```

Here each variable $\$y$ and $\$z$ stands for an *author* node. Please note, we are explicitly saying this cannot be the same node. However, such requirement cannot be expressed with our simple query language, and thus for the query *<author::Jurgen, author::Daniel>* both heuristics would return false positives. This problem can be easily verified with Ant target *test-combination-1* (see Table 2 for details).

This case raises also another problem. In the situation when single node matches all query terms, e.g. here an *author* node with *"Jurgen Daniel"* as a value, both heuristics would return not the node of publication containing this node, but the *author* node. This is because a tuple of meaningfully related nodes is build from the same node, and its LCA is the node itself. We cannot define anything like `RETURN` or `SELECT` clause for the query. We also cannot add the type of a publication (*incollection::*, *book::* etc.) to the query, because we do not know it in advance. The trick is to introduce another node to the tuple of meaningfully related nodes, by adding the query term *title::* to the query. This solution can be observed with Ant target *test-combination-2* (see Table 2 for details).

5.1.3 Sibling relationships and optional elements

Let us assume we are looking for publications written by both Jennings and Wooldridge. Precisely, the query would be formally expressed in the following way[11]:

```
for $x in collection("dblp.dbxml")/dblp/*
for $x in $x/author
for $z in $x/author
where
  functx:contains-word($y,"Jennings")
  and (functx:contains-word($z,"Wooldridge")
  and ($y != $z)
return $x
```

Let us execute this query on the data presented in Figure 13a. Here element *author*(1) and *author*(2) are *sibling*

[11]The function `functx:contains-word()` is different from default `contains()` function in that, it requires from the second argument more then being a substring of the first one – it must be a separate word.

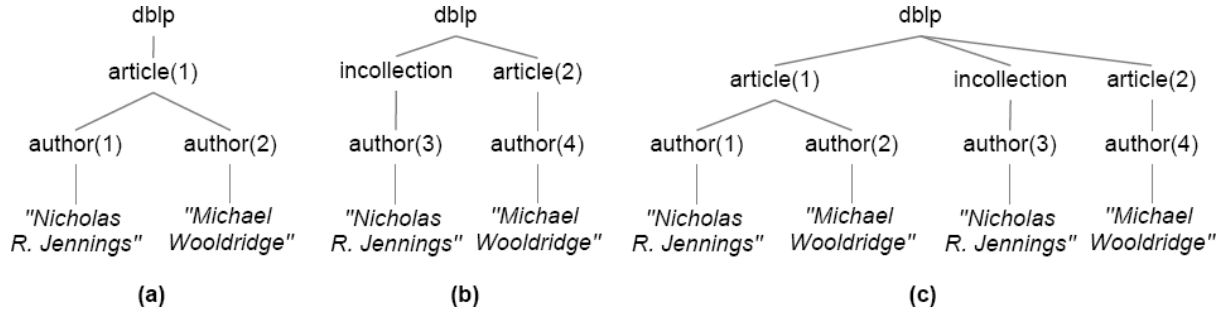

Figure 13: Examples of (a) sibling relationships, (b) optional elements and (c) sibling relationships and optional nodes together.

relationships with a common ancestor *article(1)*. In this case search engine for both heuristics would return the latter, correctly, as an answer. However, in Figure 13b we can observe the document, in which Jennings and Wooldridge are authors of two apparent publications and thus *author(3)* and *author(4)* are called *optional elements* (not meaningful). Here both heuristics does not have enough information within the document to prune redundant relationships between authors of two separate publications and thus consider {*author(1), author(2)*} as a meaningfully related tuple. Therefore our search engine would return *dblp* node as a result. However, this would not be the case for the document shown in Figure 13c. Here, our search engine for both heuristics returns *article(1)* element, but only for XRank it results in *dblp* element. This is because SLCA prunes *dblp* element, as it is ancestor of already found result *article(1)*.

5.2 Practical comparison

From Section 5.1.3 we know that the difference between both heuristics can be observed on sibling relationships and optional nodes. We verified the theory with practical experiment. We posed the query $<$*author::Jennings,author::Wooldridge*$>$ for both heuristics.

5.2.1 Measuring effectiveness

On Figure 14 we present recall-versus-precision graph for both heuristics. It can be seen that the precision is the same for all recall levels. However, from precise results we know that general precision of XRank heuristic for this query is $35/36 \approx 0.97$ (while for SLCA – 35/35), because the last position in the ranking is not relevant (it has been illustrated in Table 3). This cannot be observed on the recall-versus-precision graph, because observed points includes only these positions in the ranking, which produces an increase of recall. Since at position 35^{th} of the ranking we know that all relevant answers has been found and recall achieved a level of 100%, then position 36^{th} cannot be treated as an observation point.

Please note also, that the rank of last result is 0.4, which is lower then for all other results. This directly reflects specificity of this result, as its root node (which is also a root of the document) is far away from meaningfully related nodes[12] of the result: *article* node number 379225 and *incollection* node number 4987. It is also obvious that none of these articles can be a relevant result itself, which can be simply verified via Berkeley DB XML shell, e.g. for *incollection* node:

```
dbxml> query
  collection("dblp.dbxml")
  /dblp[1]/incollection[4987]
1 objects returned for eager expression
  'collection("text.dbxml")
    /dblp[1]/incollection[4987]'

dbxml> print
<incollection mdate="2008-03-06"
    key="series/sci/HalgamugeGJ05">
  <author>Malka Halgamuge</author>
  <author>Siddeswara Mayura Guru</author>
  <author>Andrew Jennings</author>
  <title>Centralised Strategies for Cluster
    Formation in Sensor Networks.</title>
  <pages>315-331</pages>
  <year>2005</year>
  <booktitle>Classification and Clustering for
    Knowledge Discovery</booktitle>
  <ee>http://dx.doi.org/10.1007/11011620_20</ee>
  <crossref>series/sci/2005-4</crossref>
  <url>db/series/sci/sci4.html#HalgamugeGJ05</url>
</incollection>
```

On the contrary, in the same way we can verify that the first answer is relevant:

```
dbxml> query collection("text.dbxml")
  /dblp[1]/incollection[2302]
1 objects returned for eager expression
  'collection("text.dbxml")
  /dblp[1]/incollection[2302]'

dbxml> print
<incollection mdate="2003-07-16"
    key="books/sp/omicini01/ZambonelliJOW01">
  <author>Franco Zambonelli</author>
  <author>Nicholas R. Jennings</author>
  <author>Andrea Omicini</author>
  <author>Michael Wooldridge</author>
  <title>Agent-Oriented Software Engineering for
    Internet Applications.</title>
  <pages>326-346</pages>
```

[12]Ranking method utilise positions of all matching nodes in the document, that does not belong to any other result, not only meaningfully related nodes (see Section 2.3.2). For the purpose of clarity we present here only the latter.

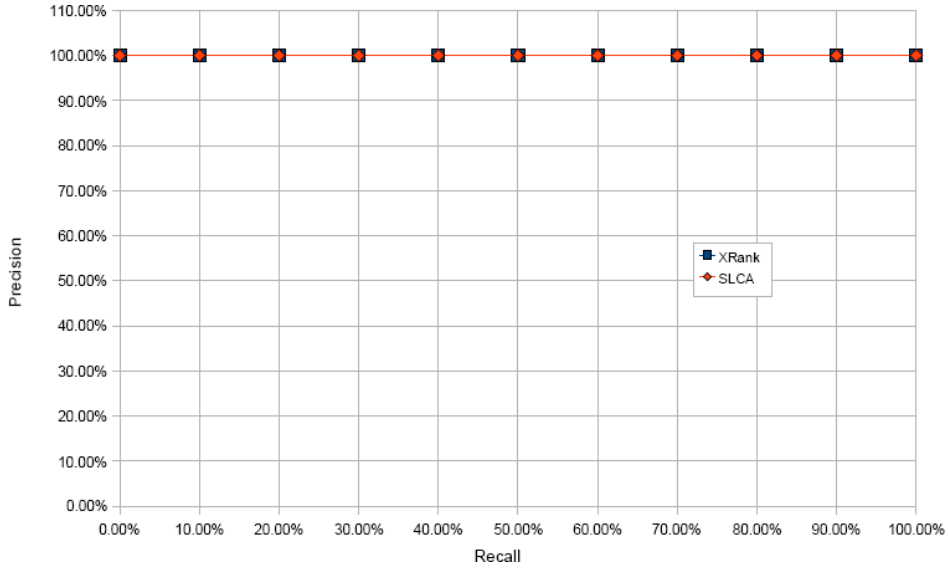

Figure 14: Recall-versus-precision graph for <*author::Jennings,author::Wooldridge*> query executed on DBLP data..

Pos.:	**Rank:**	**Relevant:**	**Document:**	**Path:**	**Row:Col**
1	1.0	true	`file:etc/data/dblp/dblp.xml`	/dblp[1]/incollection[2302]	32930:75
		Matching:	**Document:**	**Path:**	**Row:Col**
		author::Wooldridge	`file:etc/data/dblp/dblp.xml`	/dblp[1]/incollection[2302]/author[4]	32934:9
		author::Jennings	`file:etc/data/dblp/dblp.xml`	/dblp[1]/incollection[2302]/author[2]	32932:9
Pos.:	**Rank:**	**Relevant:**	**Document:**	**Path:**	**Row:Col**
...	...	...	...	...	...
Pos.:	**Rank:**	**Relevant:**	**Document:**	**Path:**	**Row:Col**
35	1.0	true	`file:etc/data/dblp/dblp.xml`	/dblp[1]/article[348165]	11400574:65
		Matching:	**Document:**	**Path:**	**Row:Col**
		author::Wooldridge	`file:etc/data/dblp/dblp.xml`	/dblp[1]/article[348165]/author[2]	11400576:9
		author::Jennings	`file:etc/data/dblp/dblp.xml`	/dblp[1]/article[348165]/author[3]	11400577:9
Pos.:	**Rank:**	**Relevant:**	**Document:**	**Path:**	**Row:Col**
36	0.4	false	`file:etc/data/dblp/dblp.xml`	/dblp[1]	3:7
		Matching:	**Document:**	**Path:**	**Row:Col**
		author::Wooldridge	`file:etc/data/dblp/dblp.xml`	/dblp[1]/article[379225]/author[3]	11793074:9
		author::Jennings	`file:etc/data/dblp/dblp.xml`	/dblp[1]/article[4987]/author[3]	11932962:9

Table 3: Excerpt of results for <*author::Jennings,author::Wooldridge*> query executed by XRank heuristic on DBLP data.

```
  <year>2001</year>
  <booktitle>Coordination of Internet Agents:
    Models, Technologies, and Applications
    </booktitle>
  <url>db/books/collections/omicini01.html
    #ZambonelliJOW01</url>
</incollection>
```

5.2.2 Measuring efficiency

We also compared efficiency of both heuristics in terms of execution time. Measured execution time includes only the retrieval process itself. Execution times of reading test collection file, ranking normalisation, measuring effectiveness and write test results to log were not concerned. Test has been performed on IBM ThinkPad R61i machine with Intel Core DUO 1.50 GHz CPU, 1 GB of memory and WD Caviar (5400 rpm) disc drive under the Windows XP system with JDK 1.6 environment. We repeated each experiment 4 times to obtain more object execution times (see Table 2 for details how to launch this experiment).

Average execution time for XRank was about 697.094 seconds, while for SLCA it was about 636.969 seconds. The difference of about 60 seconds results probably from the fact, that after finding a result node, SLCA does not "waste" time for checking whether its ancestor can be also a result[13]. This is also why SLCA is not only faster, but has also higher precision for these particular question and data.

5.3 Other test cases

5.3.1 Asynchronous execution

XML document streams can be potentially unbounded in size, thus it is not realistic to wait for results until the retrieval process ends. The solution to this problem can be asynchronous retrieval, where a process of retrieval is made in background, while the user is able to see the discovered results at any time and without interrupting retrieval process.

To simulate this case we prepared four Ant tasks presented in Table 2. In case of asynchronous execution there should appear[14] the following prompt:

```
[java] INFO: Press:
[java]  * R and ENTER to see already found results
[java]  * Ctrl+Z and ENTER to exit
```

6. CONCLUSIONS AND FUTURE WORK

Evaluation of query execution over stream data is challenging research problem. In our work we developed simple framework for measuring precision versus recall of two heuristics working on XML document stream. For this purpose we proposed solution for unique identification of result nodes among different representations of the same XML document. Our framework provides also a user with an easy way to write test configurations and read test results. We also designed and implemented simple ranking algorithm for XML document streams. Finally we compared behaviour of both heuristics on DBLP data. We have practically proved that for a specific type of questions SLCA is more precise and faster. Moreover, we proved that more differences between both heuristics can be observed on structurally richer data then DBLP. This could be one of future research directions.

Of course, there are many other interesting open questions and problem relating to our framework. The following we consider as the most important to answer in the further research and development of our framework:

- *Rationality of benchmark.* In the presented benchmark we simulated XML document streams by streaming persistent documents, designed for measuring performance of persistent data, like DBLP or XMark. In this sense our benchmark is not realistic, because real data streams comes from different sources: environmental sensor networks, network routing traces, financial transactions and cell phone call records. Moreover, measuring the length of time to execute a query is reasonable measure only for traditional DBMS benchmarks. In stream query systems, the input is continuous and theoretically querying never ends. What matters is to get answers to queries on these streams in real time or – practically – being as close to this idea as possible. By "practically" we mean having a good compromise between the state of having all relevant answers and the state of having relevant answers in real time, i.e. between effectiveness and efficiency. The authors of NEXMark [15] benchmark, stream extension to XMark benchmark [7], propose two ways of measuring these factors in case of streams: (a) *input stream rate* and *output matching* (a combined metric of output timeliness and accuracy, i.e. how actual and accurate a response is); and (b) *tuple latency* (reports how long it takes for a tuple that is relevant to the result to go through the system). The work of NEXMark includes not only performance metrics but also Firehose Stream Generator, data streams generator that models the state of an auction in XML format. The benchmark propose also various queries over the generated data. Relevant answers can be easily prepared because the stream of data generated by the generator is repetitive. However, queries of both XMark and NEXMark benchmarks are mostly oriented for aggregating numerical values and operating on them, for example one of suggested question is: *"Return the first and current increases of all open auctions whose current increase is at least twice as high as the initial increase"*. This is impossible to be expressed in syntax of our query language. Therefore some further research must be done to find or synthesise adequate benchmark.

- *Multi-word phrases.* According to the adapted algorithms for keyword matching [13], only tokenized separate words contained by XML elements are compared with a user query. Therefore it is not possible with the application to look for phrases longer then one word, like e.g. *United States*. This limitation should be carefully solved in the future.

- *Matching nodes attributes.* The application does not query against attribute names and values of XML node. This can be easily introduced in procedure handling a

[13]However precise tests would be necessary to verify how much time each heuristic spends in particular methods handling with SAX events.

[14]If no message appears in the console it is possible to make the console more verbose by setting up `.level` parameter to value `ALL` in `etc/logging.properties` file.

startElement SAX event by treating attribute name and value as a name and value of a leaf node are treated.

- *Various aggregating functions for ranking.* In Section 2.3.2 we described that ranking function computes rank for occurrences of the same query term by applying aggregate function. It would be valuable to observe how the ranking depends on the choice of aggregate function (e.g. sum, average value), especially in the situations, when one nodes matches more then one query term.
- *Lazy evaluation for asynchronous processing.* It would be interesting to provide a user also with possibility of deciding whether she wants to look for the next result, as it is done in *lazy query evaluation* approach. However, this would require *pull* stream parser, while SAX parser work in a *push* mode.
- *Appropriateness of recall/precision measures for empty sets of relevant documents.* If the set of relevant documents is empty, then it would be better to have no answers from the system at all. For example, it would be interesting to know if two scientist have written a paper together (see Section 5.1.3 for details). However, precision and recall measures does not give any penalty for false positives in this case. It would be valuable to know current solutions for this problem or appropriate alternative measures.
- *Evaluation scalability.* Finding relevant answers with `FindRelevantAnswers` tool is consuming (about 2 hours) a lot of time in comparison to the average time of processing a stream by a heuristic (about 10 minutes). This results from the fact that Berkeley DB XML database does not have a way to create a generic XPath expression from a specific node reference, because positional information is not maintained in indexes. We solved it by applying an *ad hoc* method which iterates over sibling nodes of the same name. However, it cannot be made much more efficient. On the other hand Berkeley DB XML database does have a way to generate a node handle that can be given back to the application as direct reference for but this reference will not work with other XML processors, like SAX parser.

7. ACKNOWLEDGMENTS

The author would like to thank for fruitful discussions: to Federica Mandreoli and Giorgio Villani from Information Systems Group at University of Modena and Reggio-Emilia, and to Minor Gordon from Computer Laboratory of University of Cambridge. The author thanks also to George Feinberg from Sleepycat Software and John Snelson from Oracle for help in debugging Berkeley DB XML database.

8. REFERENCES

[1] Apache Xerces Java Parser. http://xerces.apache.org/xerces-j/.
[2] DBLP, August 2008.
[3] JAXB Reference Implementation Project. https://jaxb.dev.java.net/, August 2008.
[4] JSR 222: Java Architecture for XML Binding (JAXB) 2.0, August 2008.
[5] Oracle Berkeley DB XML. http://www.oracle.com/database/berkeley-db/xml, September 2008.
[6] SAX. http://www.saxproject.org/, September 2008.
[7] XMark XML data generator. http://monetdb.cwi.nl/xml/index.html, August 2008.
[8] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, September 2008.
[9] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, September 2008.
[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.
[11] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
[12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27, New York, NY, USA, 2003. ACM.
[13] M. M. Moro and Z. Vagena. Semantic Search over XML Document Streams. In *International Workshop on Database Technologies for Handling XML Information on the Web (DATAX)*, 2008.
[14] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *Proceedings of the 17th International Conference on Data Engineering*, pages 321–329, Washington, DC, USA, 2001. IEEE Computer Society.
[15] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark A Benchmark for Queries over Data Streams (DRAFT). Technical report, OGI School of Science & Engineering at OHSU, Septembers 2008.
[16] Z. Vagena, L. S. Colby, F. zcan, A. Balmin, and Q. Li. On the effectiveness of flexible querying heuristics for xml data. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, and R. Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2007.
[17] C. J. van Rijsbergen. *Information Retrieval*, chapter Evaluation, pages 112–140. Second edition, 1999.
[18] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2005. ACM.

```
<experiments>
  <experiment rankingModel="XRankRankinizer"
    asynchronous="false"
    evaluator="XRankHeuristic"
    name="Search synchronously...">
    <execution elapsedNanoseconds="116235723"
      elapsedSeconds="0.11"
      executed="2008-09-02T20:33:25.421+02:00">
      <results sizeBeforePruning="3">
        <result isRelevant="true"
            rank="1.0">
          <root label="incollection">
            <position
              path="/dblp[1]/incollection[2]"
              row="16" col="83"
              document=
              "file:etc/data/dblp/dblp=.xml"/>
          </root>
          <meaningfullyRelated
              label="author">
            <position path=
              "/dblp[1]/incollection[2]/author[1]"
              row="17" col="13"
              document=
                "file:etc/data/dblp/dblp.xml"/>
            <matchingQueryTerm>author::Daniel
              </matchingQueryTerm>
            <matchingQueryTerm>author::Jurgen
              </matchingQueryTerm>
          </meaningfullyRelated>
          ...
        </result>
        ...
      </results>
      <extendedSummary
          generalPrecision="0.33">
        <ratio recall="100.0"
          precision="100.0"/>
      </extendedSummary>
      <standardSummary>
        <ratio recall="0.0"
          precision="100.0"/>
        <ratio recall="10.0"
          precision="100.0"/>
        <ratio recall="20.0"
          precision="100.0"/>
        ...
        <ratio recall="100.0"
          precision="100.0"/>
      </standardSummary>
    </execution>
  </experiment>
</experiments>
```

Figure 10: Example of *experiments* section in *test collection log* file.